This file consists of the second part of Chapter 2 of **MacAnova User's Guide** by Gary W. Oehlert and Christopher Bingham, issued as Technical Report Number 617, School of Statistics, University of Minnesota, March 1997, describing Version 4.04 of MacAnova

**This manual is Copyright © 1997 Gary W. Oehlert and Christopher Bingham all rights reserved.**

Fonts used in this chapter are Palatino, Courier, and Symbol.

For information concerning MacAnova, write University of Minnesota, Department of Applied Statistics, 352 Classroom Office Building, 1994 Buford Avenue, St. Paul, MN 55108-6042.

2.11 **Reading data from a file** Although you can type small data sets directly into MacAnova using `vector()` and `matrix()`, it is often more convenient to read data from a file on your hard or floppy disk. The file might have been created in a word processor such as Microsoft Word, a text editor such as Edit in DOS, or by a spread sheet or data base program.

MacAnova can read data only from plain *text* files (type `TEXT` on the Macintosh). A plain text file (sometimes called an ASCII file) is one with no additional information such as font or point size. Hence, if you use a word processor to create or edit data files to be read by MacAnova, it is *essential* that they be saved as Text or ASCII files. In some programs such as Microsoft Word, you may have to click on a **File Format** button to display choices; in others the options may be displayed in the dialog box brought up by **Save** or **Save As...**. If you do not choose a Text format, MacAnova will probably not be able to read the file.

The MacAnova functions for reading data from a plain text file are `vecread()` and `matread()`. In addition there are two pre-defined macros, `readcols` and `getdata`.

All commands which read from a file must have the file name as their first argument. This must be a quoted string or `CHARACTER` variable giving the name of the data file. In the windowed versions (Macintosh, Windows and Motif), if the file name is `""`, that is, it consists of two adjacent quotation marks, you are presented with a dialog box with a scrolling list of files from which you can select the file.

All the commands which read from a text file, including `vecread()`, `matread()` and `macroread()` can also "read" from MacAnova `CHARACTER` variables using keyword `phrase string:CharVar` as first argument instead of the file name. See Sec. 8.3.2 for details.

2.11.1 **vecread()** Function `vecread()` creates a `REAL` vector from unstructured numerical data in a file. The numbers should be separated by blanks or tab characters, or may be on separate lines. One or more successive question marks (?, ??, ???, ...) in the file is read as `MISSING` as are isolated periods (`.`) and asterisks (`*`). A exclamation point "`!`" is a stopping character and terminates the read. Any other non-numeric characters such as letters, commas or slashes will be ignored except possibly for printing an advisory message. The entire file will be scanned, unless a `!` is found. Here is a listing of a file `myfile.dat` containing 30 comma-separated values of a variable x that might be read by `vecread()`:

```
Data from trees
600, 555, 361, 489, 640,  ??  644, 297, 481, 612, 522
246, 504, 358, 623, 614, 595, 531, 602, 684, 410, 448
662, 892, 644,   ., 431, 472, 513, 603e ! 30, 10
10, 20, 3 This line will not be read
```

The output of `vecread()` is a `REAL` vector consisting of the values read.

```
Cmd> x <- vecread("myfile.dat");x # read from file myfile.dat
WARNING: nonnumeric character(s) ignored on myfile.dat
 (1)          600          555          361          489          640
 (6)      MISSING          644          297          481          612
(11)          522          246          504          358          623
(16)          614          595          531          602          684
(21)          410          448          662          892          644
(26)      MISSING          431          472          513          603
```

Note that ?? and . were read as MISSING and nothing after the "!" was read. The WARNING message results from the non-numeric first line.

You can specify a different stopping character, say "%", instead of "!" by vecread(filename,stop:"%"). In addition, if the file contains lines, all beginning with the same character, say "#", that should be ignored, you can use keyword phrase skip:"#" as an argument. Thus, suppose myfile1.dat looks like

```
# Data on 30 trees
600, 555, 361, 489, 640,   ?  644, 297, 481, 612, 522
246, 504, 358, 623, 614, 595, 531, 602, 684, 410, 448
662, 892, 644,   .  431, 472, 513, 603 % 30, 10
10, 20, 3 This line will not be read
```

Then vecread("myfile1.dat",stop:"%",skip:"#") will return exactly the same vector as before. If the first line were not skipped, the 30 would be read as an item of data.

Suppose myfile2.dat contains data arranged in columns like the following

```
# Data on 5 variables, n = 6
   600   555   361   489   640
     ?   644   297   481   612
   522   246   504   358   623
   614   595   531   602   684
   410   448   662   892   644
     ?   431   472   513   603
```

You can read it into matrix y as follows:

```
Cmd> y <- matrix(vecread(myfile2.dat,skip:"#",quiet:F),5)'
# Data on 5 variables, n = 6   Echoed because of quiet:F

Cmd> y
(1,1)          600          555          361          489          640
(2,1)      MISSING          644          297          481          612
(3,1)          522          246          504          358          623
(4,1)          614          595          531          602          684
(5,1)          410          448          662          892          644
(6,1)      MISSING          431          472          513          603
```

Note keyword phrase quiet:F which results in any skipped lines being printed. Transposing the output of matrix() is necessary because data is read from the file row by row, but stored in the computer column by column. Thus simply y <- matrix(vecread("myfile2.dat",skip:"#"),5) would produce a matrix with 5 rows, with each row corresponding to a column of the data in the file.

You can also use vecread() to read CHARACTER data (see also Sec. 7.2). Suppose file labels.txt contains

```
Age, Length, Height, Width,   , Strength ! labels for data
```

Then vecread() reads a vector of length 6:

```
Cmd> labels <- vecread("labels.txt",char:T); labels
(1) "Age"
(2) "Length"
(3) "Height"
(4) "Width"
(5) ""
(6) "Strength"
```

Note that two successive commas are taken to delimit an empty string.

You can use keyword phrase silent:T to suppress any warning messages. Thus if the file starship.txt looks like

```
Troy     342 67
Tasha    546 53
Beverly  331 49
```

```
Cmd> y <- matrix(vecread("starship.txt",silent:T),2)';y
(1,1)          342           67
(2,1)          546           53
(3,1)          331           49
```

returns the 3 by 2 data matrix without commenting on the row labels.

An alternative method of coping with unreadable items is to use keyword phrase badvalue:val, where val is a REAL scalar or ?. Any item that does not appear to be a number is read as if it had this value.

```
Cmd> y <- matrix(vecread("starship.txt",badvalue:-99),3)';y
(1,1)          -99          342           67
(2,1)          -99          546           53
(3,1)          -99          331           49
```

The three names could not be read and were replaced by -99. This has the advantage that you can test to see which items were not readable.

See Sec. 7.3 for information on using vecread() to read CHARACTER data.

2.11.2 **readcols** When a data file such as myfile2.dat contains data on several variables, the data for each case on a separate line, you sometimes want to read each variable (column in the file) into a separate MacAnova variable. You sometimes can use pre-defined macro readcols, which makes use of vecread(), to read such file and create REAL vectors, one for each column in the file. The arguments to readcols are the file name, and the unquoted names of the variables into which the columns should be placed.

```
Cmd> readcols("myfile2.dat", x1, x2, x3, x4, x5,skip:"#")
# Data on 5 variables, n = 6
```

This creates REAL vectors x1, x2, x3, x4, and y containing the data from columns 1 through 5.

```
Cmd> list(x1,x2,x3,x4,x5)
x1              REAL   6
x2              REAL   6
x3              REAL   6
x4              REAL   6
x5              REAL   6

Cmd> x1 # column (variable) 1
(1)          600      MISSING          522          614          410
(6)      MISSING
```

If the file has a different number of columns from what you expect,readcols usually prints an error message.

```
Cmd> readcols("myfile2.dat",x1, x2, x3, x4,skip:"#")# try 4 cols
# Data on 5 variables, n = 6
ERROR: number of rows must divide length of data
```

However, if the number of data elements in the file is actually divisible by the number of variable names you provided, there will be no error message, but the variables set will be of the wrong length and will not correspond to columns in the file.

```
Cmd> readcols("myfile2.dat", x1, x2, x3, x4, x5, x6,skip:"#")
# Data on 5 variables, n = 6

Cmd> list(x1)
x1              REAL   5    Output from list

Cmd> x1 # x1 has length 5, not 6
(1)          600          644          504          602          644
```

Macro readcols recognizes vecread() keywords stop, skip, quiet, silent and badvalue. See Sec. 2.11.1 for details.

2.11.3 **matread()** A single plain text file can contain several REAL, LOGICAL, or CHARACTER data sets, provided each of them has a *header* consisting of at least one line of information specifying a *name* for the data set and its *dimensions* (length for a vector, numbers of rows and columns for a matrix). Optional additional header lines can describe the data set and provide formatting information. It can also contain NULL variables and structures. See Sec. 7.1 for details of the file format. You use matread() to read from such a file.

Function matread() normally requires two CHARACTER variables or quoted strings as arguments, the file name and the data set name. Thus

```
Cmd> x <- matread("data.txt","treedata")
```

would search file data.txt for a data set named treedata, read it, and assign the data variable x. If you omit the data set name, matread() assumes that the first non-empty line in the file is the first header line for the data set. You can create files in the format readable by matread() using matprint() and matwrite() (see Sec. 7.4). If the named file is not in the current default directory or folder, matread() searches the directories

or folders in CHARACTER vector DATAPATHS. See Sec. 2.11.6.

File MacAnova.dat distributed with MacAnova contains the data in a form readable by matread(). One of the data sets is halddata, containing data that has often been used as an example when demonstrating regression techniques.

```
Cmd> hald <- matread("macanova.dat","halddata")
halddata     13      5 format
) Hald data from A. Hald, Statistical Theory with Engineering
) Applications, Wiley, New York, 1952, p. 647
) Col. 1: X1 = percent tricalcium aluminate
) Col. 2: X2 = percent tricalcium silicate
) Col. 3: X3 = percent tetracalcium alumino ferrite
) Col. 4: X4 = percent dicalcium silicate
) Col. 5: Y  = cumulative heat evolved from cement hardening after
)              180 days. (calories/gm)
```

The lines printed by matread() are the name line and several additional header lines starting with ")" (comment lines) which come before the data. You can suppress the printing of these lines by including quiet:T as an additional argument to matread().

Since some commands work only with data vectors rather than matrices, once you have read in a matrix using matread(), you may want to split it up into separate vectors, each containing a column of the matrix. Pre-defined macro makecols does exactly this. For example, you can create vectors from data by

```
Cmd> makecols(hald,x1,x2,x3,x4,y)
```

will create vectors x1, x2, x3, x4 and y from the 5 columns of hald.

You can even combine matread() and makecols in a single expression to create variables.

```
Cmd> makecols(matread("macanova.dat","halddata",quiet:T),\
x1,x2,x3,x4,y)
```

Ordinarily, it is considered an error when matread() cannot find a data set, this is considered to be an error. If you include keyword phrase notfoundok:T as an argument, no error will be reported and the value NULL is returned. This can be useful in writing macros (see Sec. 9.3) since it allows you to test whether a data set is available on a file.

```
Cmd> x <- matread("macanova.dat","appledata") # no appledata in file
ERROR: dataset or macro appledata not found on file macanova.dat

Cmd> print(x)
ERROR: argument 1 (x) to print is undefined

Cmd> x <- matread("macanova.dat","appledata",notfoundok:T)

Cmd> print(x)
x:
(NULL)
```

Coordinate labels (see Sec. 8.4) are not part of the file format of data sets readable by matread() (See. Sec. 7.1) and thus vectors, matrices and arrays read by matread() do not normally have such labels. You can, however, add labels using matread()

keyword labels. Here is an example, reading halddata; echoing of the header lines is suppressed by quiet:T:

```
Cmd> data <- matread("macanova.dat","halddata",quiet:T,\
labels:structure("Obs. ", vector("TricalcAlum","TricalcSi",\
"TetrcalAlFe","DicalcSi","CumulHeat")))

Cmd> print(data,nsig:4) # nsig:4 means print 4 signif. digits
data:
        TricalcAlum   TricalcSi TetrcalAlFe    DicalcSi   CumulHeat
Obs. 1            7           26           6          60        78.5
Obs. 2            1           29          15          52        74.3
Obs. 3           11           56           8          20       104.3
Obs. 4           11           31           8          47        87.6
Obs. 5            7           52           6          33        95.9
Obs. 6           11           55           9          22       109.2
Obs. 7            3           71          17           6       102.7
Obs. 8            1           31          22          44        72.5
Obs. 9            2           54          18          22        93.1
Obs. 10          21           47           4          26       115.9
Obs. 11           1           40          23          34        83.8
Obs. 12          11           66           9          12       113.3
Obs. 13          10           68           8          12       109.4
```

2.11.4 **getdata** Macro getdata is designed make it easier to work with "libraries" of data sets in the form readable by matread(). It has a single *unquoted* argument that is interpreted as the name of a data set on the file whose name is the value of CHARACTER variable DATAFILE. When MacAnova is started up, DATAFILE is pre-defined to be "macanova.dat", but you may assign a new value at any time. getdata uses matread() to read the file. For example, using the default value of DATAFILE,

```
Cmd> hald <- getdata(halddata)
```

would be equivalent to hald <- matread("macanova.dat","halddata"). To use getdata to retrieve treedata from file data.txt, you would need the following:

```
Cmd> DATAFILE <- "data.txt"; x <- getdata(treedata)
```

Subsequent use of getdata would continue to retrieve data from data.txt.

If the file named in DATAFILE is not in the default directory or folder or in one of the directories or folders specified by CHARACTER vector DATAPATHS (see Sec. 2.11.6), it should be a complete "path name", such as "C:/MACANOVA/MACANOVA.DAT" or "Macintosh Hard Disk:MacAnova:MacAnova.dat". See Appendices B through F. Note that on DOS/Windows computers you can use "/" instead of "\" in path names. In fact, if you want to use "\" you have to use "\\" (see Sec. 2.5).

2.11.5 **inforead()** As exemplified in Sec. 2.11.3, data sets on external files that are readable by matread() may have associated comment lines that give information about the data (see also Sec. 7.1). The same is true of macros on external files (Sec. 7.5). It can be helpful to save these comments in a CHARACTER variable so that they are instantly available for references. You can do this using inforead() which is used much the same as matread().

inforead(FileName,Name) searches file FileName for a macro or data set with name

Name. If found, it reads the comments (the lines starting with ")") following the header line and returns a CHARACTER variable containing these lines, with the leading ")" stripped off. FileName and Name must be quoted strings or CHARACTER variables. The actual contents of the data set or macro are ignored and there is no checking as to whether the header line is in correct format.

inforead(FileName) does the same for the first data set or macro on the file, assuming that line 1 is the header line.

In versions with windows (Macintosh, Windows, Motif), if FileName is the null string "", you will be able to select the file using a dialog box.

inforead(FileName[,Name],quiet:T) returns the comments, but suppresses any echoing of the header and comment lines of the data set or macro.

```
Cmd> info <- inforead("macanova.dat","halddata",quiet:T)

Cmd> info # see Sec. 2.11.3.
(1) " Hald data from A. Hald, Statistical Theory with Engineering
 Applications, Wiley, New York, 1952, p. 647
 Col. 1: X1 = percent tricalcium aluminate
 Col. 2: X2 = percent tricalcium silicate
 Col. 3: X3 = percent tetracalcium alumino ferrite
 Col. 4: X4 = percent dicalcium silicate
 Col. 5: Y  = cumulative heat evolved from cement hardening after
              180 days. (calories/gm)"
```

inforead(FileName,Name[,quiet:T],notfoundok:T) behaves identically if the data set or macro is found. However, if it is not, no message is printed and NULL is returned. When used in a macro, this feature allows special action if Name is not found.

```
Cmd> appleinfo <- matread("macanova.dat","appledata",notfoundok:T)

Cmd> print(appleinfo)
appleinfo:
(NULL)
```

### 2.11.6 **HOME, DATAPATHS and adddatapath**

When MacAnova is started up, two CHARACTER variables, HOME and DATAPATHS, are created. On all systems except Unix, these are both initialized to the complete "path name" of the directory or folder where MacAnova is located. On a Unix system, HOME is initialized to the name of the user's home directory (environmental variable $HOME) and DATAPATHS is initialized to the name of an installation-dependent directory containing data files. Both variables can be modified, if desired, once MacAnova has started up, perhaps in a start up file (See Sec. 7.8). In particular, you can add other directory or folder names to DATAPATHS, making it a CHARACTER vector. HOME and DATAPATHS make it easier for MacAnova to find files.

If a file name is specified as a simple file name, such as "halddata", with no special path characters such as ":" or "/", Macanova first attempts to read it in whatever the current default directory or folder is. If it is not successful, then it makes an attempt in directory DATAPATHS[1]; if not successful there, it looks in DATAPATHS[2], and so on, giving up only if it is not found in any directory or folder in DATAPATHS. Here is an example of its use. File Hald is not in the default directory/folder or in DATAPATHS[1]

but is in a sub-directory or folder.

```
Cmd> DATAPATHS # one folder name in DATAPATHS
(1) "Macintosh HD:MacAnova Folder:"

Cmd> y <- vecread("Hald") # can't find file Hald there
ERROR: vecread cannot open file Hald

Cmd> DATAPATHS <- vector("Macintosh HD:MacAnova Folder:Data",\
DATAPATHS) # add new folder at start of DATAPATHS

Cmd> y <- vecread("Hald"); y[run(5)] # Found
(1)           7           26            6           60         78.5
```

Vector DATAPATHS can be of any length, providing lots of places to look for a file. Pre-defined macro adddatapath makes it easier to add a folder or directory name to either the beginning or end of DATAPATHS

```
Cmd> adddatapath("timeser"); DATAPATHS # add "timeser" at start
(1) "timeser"
(2) "Macintosh HD:MacAnova Folder:Data:"
(3) "Macintosh HD:MacAnova Folder:"

Cmd> adddatapath("mvdata",T);DATAPATHS # add "mvdata" at end
(1) "timeser"
(2) "Macintosh HD:MacAnova Folder:Data:"
(3) "Macintosh HD:MacAnova Folder:"
(4) "mvdata"
```

The above uses the Macintosh way of separating names in a path using ":". On other computes you would use names like "D:/MACANOVA/DATA/" (DOS/Windows) or "/users/kb/macanova/data/" (Unix).

Variable HOME comes into play when you use a file name of the form "~:name" (on a Macintosh) or "~/name" (on other computers). MacAnova looks for the file in the directory whose name is in HOME.

```
Cmd> HOME
(1) "Macintosh HD:MacAnova Folder:"

Cmd> y <- vecread("~:Data:Hald") # found
```

You can change HOME to be the name of any folder or directory.

```
Cmd> HOME <- "Macintosh HD:MacAnova Folder:Data:"

Cmd> y <- vecread("~:Hald") # found
```

2.12 **Simple statistics** While MacAnova is oriented towards the analysis of variance, multivariate analysis and time series analysis, it provides many simpler statistical functions as well.

2.12.1 **describe()** When you want simple descriptive statistics of data in variable x, use describe(x). This computes, for each column of x, the number of non-missing values, the mean, variance, median, maximum, minimum, and upper and lower quartiles.

The output of describe() is a structure, with components n, min, q1, median, q3, max,

mean, and var.

```
Cmd> x<-matrix(vector(2,?,4,5,8,2,3,4,1,7,8,4,3,6,5),5); x
(1,1)            2           2           8
(2,1)      MISSING           3           4
(3,1)            4           4           3
(4,1)            5           1           6
(5,1)            8           7           5
```

```
Cmd> describe(x)
WARNING: missing values in input to describe
WARNING: output reflects nonmissing data only
component: n       Number of non-missing values
(1)            4           5           5
component: min     Minimum value
(1)            2           1           3
component: q1      Lower quartile
(1)            3           2           4
component: median  2nd quartile
(1)          4.5           3           5
component: q3      Upper quartile
(1)          6.5           4           6
component: max     Maximum value
(1)            8           7           8
component: mean    Average
(1)         4.75         3.4         5.2
component: var     Variance
(1)         6.25         5.3         3.7
```

If you don't want all the statistics, you can specify the ones you want using keywords matching the component names.

```
Cmd> describe(x,mean:T,var:T)
WARNING: missing values in input to describe
WARNING: output reflects non-missing data only
component: mean
(1)         4.75         3.4         5.2
component: var
(1)         6.25         5.3         3.7
```

If you specify only one statistic (describe(x,mean:T)) you get a scalar or a vector, not a structure.

You can also use keyword phrase all:T as in describe(x,all:T,mean:F,var:F) which computes all the statistics except the mean and variance.

If x is a structure, describe(x) computes the statistics for each component. In that case, each component of describe(x) is itself a structure similar to x:

```
Cmd> describe(temperatures, mean:T,var:T)
component: mean
  component: Saturday
(1)         69.3
  component: Sunday
(1)           67
  component: Monday
(1)       66.786
```

```
component: var
  component: Saturday
(1)       14.233
  component: Sunday
(1)         39.5
  component: Monday
(1)       19.412
```

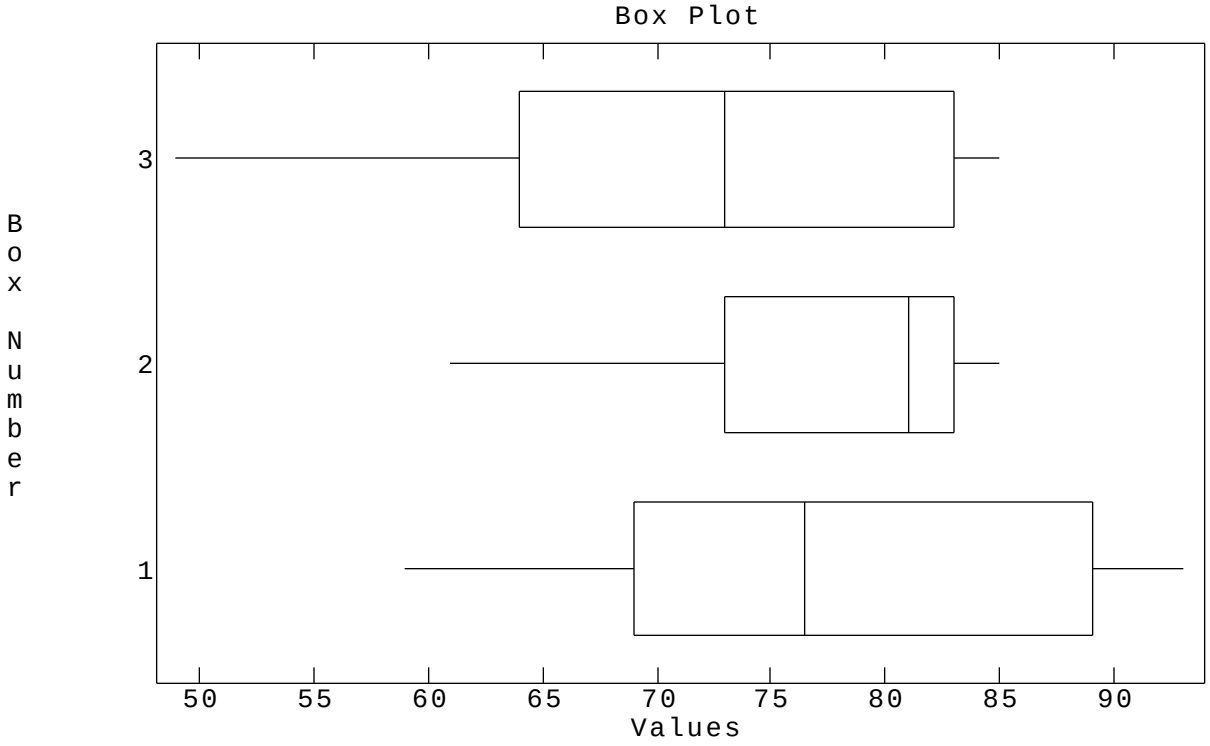
Here `temperatures` is the structure used as an example in Sec. 2.8.16.

2.12.2 **boxplot(), vboxplot, stemleaf() and hist** These are intended to give a quick look at the distribution of a sample or samples of data.

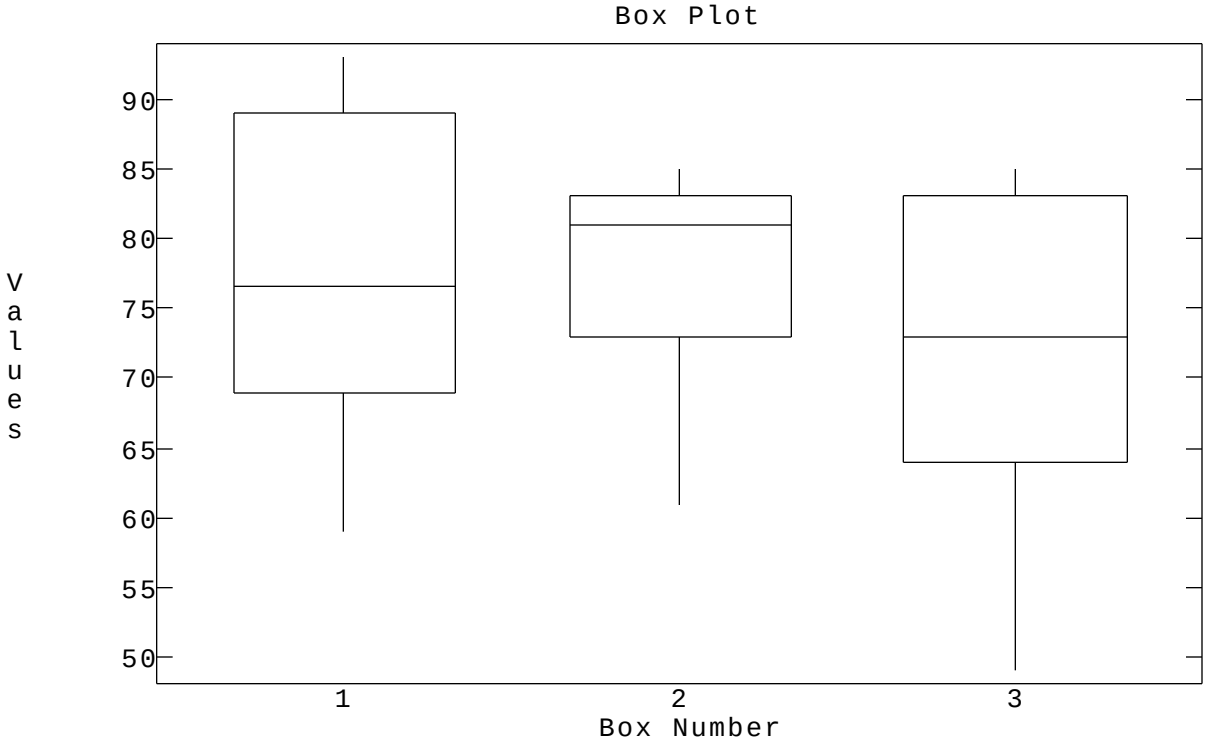
`boxplot(x)` predefined macro `vboxplot` draw a Tukey box and whisker diagram, often called a *box plot*, of the data in `x`. This is a graphical summary of the distribution of the values in `x`. When `x1`, `x2`, ..., `xk` are several variables, `boxplot(x1,x2,...,xk)` draws parallel horizontal box plots, a very useful way to compare the distributions of the different variables. `vboxplot(x1,x2,...,xk)`If you prefer a plot box plots in which the boxes are oriented vertically, use `boxplot(x,vertical:T)` or more simply, `vboxplot(x1,x2,...,xk)`.

A box plot consists of a central box, with a line drawn somewhere between the left (lower) and right (upper) ends, "whiskers" extending off the ends of the box, and separately plotted "outliers". The ends of the box are drawn at the lower and upper quartiles, respectively, and the middle line is the median. The whiskers extend to the furthest (lowest and highest) points that are still within the "inner fences" (lower quartile – 1.5 IQR and upper quartile + 1.5 IQR). Points outside the inner fences may be tentatively identified as outliers. Points outside the the "outer fences" (lower quartile - 3 IQR and upper quartile + 3 IQR) are plotted as "`o`" and and points between the inner and outer fences are plotted as "`*`".

```
Cmd> boxplot(temperatures$Saturday, temperatures$Sunday,\
temperatures$Monday)
```




```
Cmd> vboxplot(temperatures$Saturday, temperatures$Sunday,\
temperatures$Monday) # same with vertical orientation
```



The bottom or left box represents the Saturday temperatures, the middle box, the Sunday temperatures, and so on. No temperatures were outliers so there are no symbols beyond the whiskers.

```
Cmd> boxplot(temperatures$Saturday, temperatures$Sunday,\
temperatures$Monday,vertical:T)
```

is an alternative way to get vertically oriented boxes.

If `str` is a structure whose components are REAL vectors, `boxplot(str)` draws parallel box plots for each component. Thus, since `temperatures` is a structure, `boxplot(temperatures)` would produce the same display of box plots.

A common way to create a structure for use with `boxplot()` is to use `split()` (see Sec. 9.1.1). In the following, `temps` is a vector combining the data from the three components of `temperatures`, and a `day` is a vector of integers coding Saturday as 1, Sunday as 2 and Monday as 3

```
Cmd> print(temps,day,format:"2.0f") #see Sec. 7.4 for use of format
y:
 (1) 65 71 75 86 91 93 89 78 69 59 61 73 85 83 81 51 65 71 78 83 84
(22) 85 84 81 75 69 64 59 49
day:
 (1)  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  3  3  3  3  3  3
(22)  3  3  3  3  3  3  3  3
```

```
Cmd> split(temps,day) # create structure, splitting up temps by day
component: day1
(1)           65           71           75           86           91
(6)           93           89           78           69           59
component: day2
(1)           61           73           85           83           81
component: day3
 (1)           51           65           71           78           83
 (6)           84           85           84           81           75
(11)           69           64           59           49
```

```
Cmd> boxplot(split(temps,day))
```

This would produce the same horizontally oriented plot as before.

You can print a stem and leaf display of the data in a REAL vector or a n by 1 matrix `x` by `stemleaf(x)`:

```
Cmd> stemleaf(temperatures$Monday)
    1    4.|9
    2    5*|1
    3    5.|9
    4    6*|4
    6    6.|59
    7    7*|1
    7    7.|58
    5    8*|1344
    1    8.|5

         1*|1  represents 11  Leaf digit unit = 1
```

The numbers to the left of "|" are the "stems" are represent the first digit (sometimes 2 digits) of each number. The numbers to the right of "|" are "leaves" and represent the next digit of each number after the stem. Because the values of Monday temperatures range only from 49 to 85, here MacAnova uses "half digit" stems whose leaves are either 0, 1, 2, 3 or 4 (for stems labelled 5*, 6*, ...) or 5, 6, 7, 8 or 9 (for stems labelled 4., 5., ...).

You can control the number of stems stemleaf() will use by an optional second argument which allows you to set the maximum number of stems.

```
Cmd> stemleaf(temperatures$Monday,5) # use at most 5 stems
    1     4|9
    3     5|19
    6     6|459
  ( 3)    7|158
    5     8|13445

          1|1  represents 11  Leaf digit unit = 1
```

The first column printed contains the "depth" – cumulative counts from each "tail" of the distribution, plus, in parentheses, the number of leaves on the stem which includes the median. Thus, there are 5 cases with values 80 and above, 6 with values less than 70 and 3 values in the 70's. The leaves are always computed by rounding toward 0, so that a temperature of 79.9 would show up as a leaf of 9 on stem 7 and would not be rounded up to 80 (leaf of 0 on stem of 8).

When there are outliers, defined as values beyond a box plot's "inner fences", stemleaf() normally lists them separately unless outliers:F is an argument. Let's make the first Monday temperature an outlier.

```
Cmd> monday<-temperatures$Monday

Cmd> monday[1]<-120;stemleaf(monday,5)
    1     4|9
    2     5|9
    5     6|459
  ( 3)    7|158
    6     8|13445
 High  120
          1|1  represents 11  Leaf digit unit = 1
```

Keyword phrase depth:F suppresses printing the depth column and keyword phrase stat:T prints the extremes and the quartiles:

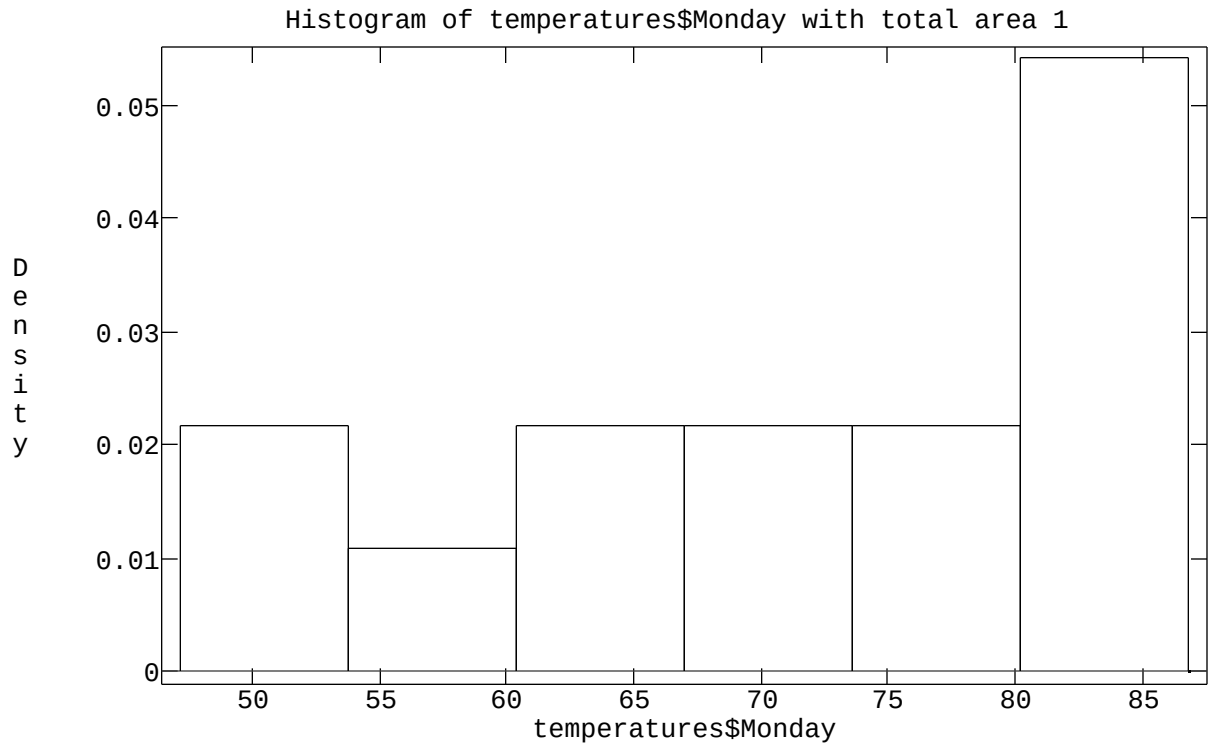
```
Cmd> stemleaf(temperatures$Saturday,5,depth:F,stat:T)
n=10, Min=59, Q1=69, M=76.5, Q3=89, Max=93
    5|9
    6|59
    7|158
    8|69
    9|13

    1|1 represents 11  Leaf digit unit = 1
```

A histogram is more conventional way to display the distribution of data points. You can draw histograms using pre-defined macro hist. If you are satisfied to let

MacAnova pick the number of bars in the histogram, you can simply type `hist(x)`, where `x` is a REAL vector. Like `stemleaf()`, an optional second argument specifies the number of bars. The heights of the bars is in the so called *density scale* -- that is the product of the width of a bar by its height is the proportional of values falling in the bar.

```
Cmd> hist(temperatures$Monday,6) # draw histogram with 6 bars
```




You can, in fact, completely specify the class limits. For example,

```
Cmd> hist(temperatures$Monday,vector(40,50,60,70,80,85,90))
```

draws a histogram with unequally spaced class limits 40, 50, 60, 70, 80, 85 and 90.

You can specify a title and X-axis labels for a histogram or for a box plot using keywords `title` and `xlab`. Examples might be

```
Cmd> boxplot(temperatures,\
title:"Saturday through Monday Temperatures",\
xlab:"Degrees Fahrenheit")

Cmd> hist(temperatures$Monday,vector(40,50,60,70,80,85,90),\
title:"Monday temperatures",xlab:"Degrees Fahrenheit")
```

Normally, `hist` considers an upper or right class limit to be in a class and the lower or left class limit to be in the next lower class. If you include `leftendin:T` as an argument to `hist`, the lower limit is in the class and the upper limit is in the next higher class. This corresponds to the convention used in some statistics text books.

2.12.3 **sort(), rank(), grade(), rankits(), and halfnorm()** These functions all have to do with the ordering of data values and expect a single vector or matrix argument plus an optional keyword phrase `down:T`. All work with REAL data; `sort()`, `rank()` and

grade() also work with CHARACTER data.

For CHARACTER data, ordering is in alphabetical order using the ASCII collating sequence in which most punctuation and all numerals sort ahead of upper case letters which sort ahead of lower case letters. A space sorts ahead of all printable characters. Here is the explicit ordering starting with space:

| Sorting order of characters |
|---|
| !"#$%&'()*+,-./0123456789:;<=>? |
| @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_ |
| `abcdefghijklmnopqrstuvwxyz{\|}~ |

A null string "" sorts before any other string.

sort(x) returns a vector or matrix, each column of which is the corresponding column of x arranged in increasing or alphabetical order. sort(x,down:T) orders each column in decreasing or reverse alphabetical order. In both cases, x is REAL, MISSING values are moved to the end.

rank(x) computes the ranks of the non-missing data in each column of x, with the smallest or alphabetically first value assigned rank 1. rank(x,down:T) does the same, except the largest or alphabetically last value assigned rank 1. For REAL data, the rank of a MISSING value is set MISSING.

By default, when there are ties in REAL data, the average of the ranks associated with each set of ties is normally computed. For CHARACTER data, the ranks associated with tied elements are distinct and unpredictable. See below for alternative treatment of ties with REAL data.

grade(x) computes what might be called inverse ranks of the columns of x. When x is a vector, after j <- grade(x), j[1] is the index (row number) of the smallest or alphabetically first element of x, j[2] is the index of the second smallest, and so on. If length(x) is n, then grade(x)[n] is the index of the largest number (maximum or last in alphabetical order). Thus x[grade(x)] is the same as sort(x). x[grade(x[,1]),] returns a matrix the same shape as x containing the rows of x rearranged so that the first column is in increasing order. This is probably the most important application of grade(). grade(x,down:T) does the same, except, j[1] is the index of the largest value or last in alphabetical order. When x is a matrix, grade(x) processes each column separately. If x is REAL, the indices of any MISSING values are always put in the last rows of grade(x) and grade(x,down:T).

Function rankits() computes the *normal scores* or *rankits* of the non-missing elements in each column and sets the rankit of a MISSING value to MISSING. When x is a vector and there are no ties, rankits(x)[i] is the normal probability point $z_{p_i}$, where $p_i$ = (rank(x)[i] - .375)/(n + .25). When there are ties, rankits() breaks them arbitrarily. Note that the order of the elements of x is retained.

Function halfnorm() is similar to rankits() except it computes *half normal scores*, that is, when there are no ties, halfnorm(x)[i] is $z_{p_i}$ where

$$p_i = \texttt{.5 + .5*(rank(abs(x))[i] - .375)/(n + .25))}.$$

```
Cmd> x<-matrix(vector(2,?,4,5,8, 2,3,4,1,7, 8,4,3,6,5),5); x
(1,1)            2            2            8
(2,1)      MISSING            3            4
(3,1)            4            4            3
(4,1)            5            1            6
(5,1)            8            7            5

Cmd> print(sort:sort(x),rank:rank(x),rankits:rankits(x),\
grade:grade(x))
WARNING:  MISSING values in argument to sort
WARNING:  MISSING values in argument to rank
WARNING:  MISSING values in argument to rankits
WARNING:  MISSING values in argument to grade
sort:
(1,1)            2            1            3
(2,1)            4            2            4
(3,1)            5            3            5
(4,1)            8            4            6
(5,1)      MISSING            7            8  MISSING at end
rank:
(1,1)            1            2            5
(2,1)      MISSING            3            2
(3,1)            2            4            1
(4,1)            3            1            4
(5,1)            4            5            3
rankits:
(1,1)      -1.0491      -0.4972       1.1798
(2,1)      MISSING            0      -0.4972
(3,1)     -0.29931       0.4972      -1.1798
(4,1)      0.29931      -1.1798       0.4972
(5,1)       1.0491       1.1798            0
grade:
(1,1)            1            4            3
(2,1)            3            1            2
(3,1)            4            2            5
(4,1)            5            3            4
(5,1)            2            5            1
```

Note the use of *keywords* `sort`, `rank`, `rankits`, and `grade` in `print()` to provide labels for the output (See. Sec. 7.4.1). Without them, all four matrices would be labeled `MATRIX`.

```
Cmd> print(downsort:sort(x,down:T),downrank:rank(x,down:T),\
downgrade:grade(x,down:T)) # same with down:T
WARNING:  MISSING values in argument to sort
WARNING:  MISSING values in argument to rank
WARNING:  MISSING values in argument to grade
downsort:
(1,1)            8            7            8
(2,1)            5            4            6
(3,1)            4            3            5
(4,1)            2            2            4
(5,1)      MISSING            1            3
```

```
downrank:
(1,1)            4            4            1
(2,1)      MISSING            3            4
(3,1)            3            2            5
(4,1)            2            5            2
(5,1)            1            1            3
downgrade:
(1,1)            5            5            1
(2,1)            4            3            4
(3,1)            3            2            5
(4,1)            1            1            2
(5,1)            2            4            3
```

With any of these functions, if x is an array, the result is an array of the same size and shape, with the function being applied to every vector specified by fixed values of subscripts 2, 3 ... .

```
Cmd> ary <- array(vector(27.3,20.8,32.0, 28.8,18.9,28.1,\
22.9,32.4,32.1,  31.4,29.1,31.3), 3, 2, 2)# 3 by 2 by 2

Cmd> ary[,2,2]
(1,1,1)         31.4
(2,1,1)         29.1
(3,1,1)         31.3

Cmd> sort(ary)[,2,2] # same as sort(ary[,2,2])
(1,1,1)         29.1
(2,1,1)         31.3
(3,1,1)         31.4
```

It is also acceptable for x to be a structure (Sec. 2.8.16), whose non-structure components are all REAL or all CHARACTER. In that case, each function returns a structure of the same form, each of whose non-structure components is the result of applying the function to the corresponding component of x.

```
Cmd> sort(structure(dry:vector(3.32,2.99,1.61,2.52),\
wet:vector(3.60,4.21,3.636)))
component: dry
(1)          1.61          2.52          2.99          3.32
component: wet
(1)           3.6         3.636          4.21
```

You can modify the handling of ties in by rank(), rankits(), and halfnorm() using keyword phrase ties:method, where method is one of "average", "minimum" or "ignore" which you can abbreviate as "a", "b" and "i". Suppose $k$ elements in a vector (column) are tied, that is they all have the same value and no other element has this value, and suppose the ranks these elements would have if their values were very slightly changed so as to break the ties while preserving other ordering would be $r$, $r$+1, $r$+2, ..., $r$+$k$–1. The following table describes the ranks computed by rank() for the tied values for each of the three possible methods.

| Value for ties | Computed ranks |
|---|---|
| "average" | All ranks = $(r+(r+1)+...+(r+k-1))/k = r + (k-1)/2$ |
| "minimum" | All ranks = $r$ |
| "ignore" | $r$, $r+1$, $r+2$, ..., $r+k-1$ in an unpredictable order |

rankits(x,ties:method) and halfnorm(x,ties:method) are equivalent to finding the normal probability points corresponding to $p_i$ = (rank(x,ties:method)[i] - .375)/(n + .25) and $p_i$ = .5 + .5*(rank(abs(x),ties:method)[i] - .375)/(n + .25)), respectively. It is hard to imagine a situation in which ties:"minimum" would be useful with rankits() and halfnorm().

```
Cmd> rank(vector(3,2,2,1,3),ties:"average") # default
(1)          4.5          2.5          2.5            1          4.5

Cmd> rank(vector(3,2,2,1,3),ties:"minimum")
(1)            4            2            2            1            4

Cmd> rank(vector(3,2,2,1,3),ties:"ignore")
(1)            4            2            3            1            5

Cmd> rankits(vector(3,2,2,1,3),ties:"average")
(1)      0.79164     -0.24104     -0.24104      -1.1798      0.79164

Cmd> rankits(vector(3,2,2,1,3),ties:"ignore") # default
(1)       0.4972      -0.4972            0      -1.1798       1.1798
```

2.12.4 **sum(), prod(), max(), min()** These functions compute summary values for a REAL or LOGICAL vector or for each column of a REAL or LOGICAL matrix. For LOGICAL data, True is interpreted as 1 and False as 0.

```
Cmd> print(sum:sum(x),prod:prod(x),min:min(x),max:max(x))
WARNING: MISSING values found by sum
WARNING: MISSING values found by prod
WARNING: MISSING values found by min
WARNING: MISSING values found by max
sum:
(1,1)           19           17           26 Sums down columns
prod:
(1,1)          320          168         2880 Product down cols
min:
(1,1)            2            1            3 Minimum value in col
max:
(1,1)            8            7            8 Maximum value in col

Cmd> sum(x <= 4) # count the numbers <= 4 in each column
WARNING: comparison with missing value(s) near sum(x <= 4)
WARNING: MISSING values found by sum
(1,1)            2            4            2
```

When there are no missing values, sum(x)/nrows(x) computes the column means of a REAL matrix x, producing a row vector.

With any of these functions, if the argument is an array, the result is an array with the

same number of dimensions but with the first dimension having length 1. The function is applied to every vector specified by fixed values of subscripts 2, 3 ... . We illustrate this use with the array `ary` created in Sec. 2.12.3.

```
Cmd> sum(ary) # sum over first dimension.
(1,1,1)         80.1         87.4
(1,2,1)         75.8         91.8

Cmd> max(ary) # maximimum over first dimensions
(1,1,1)           32         32.4
(1,2,1)         28.8         31.4
```

These functions also accept a list of `REAL` or `LOGICAL` scalars or vectors as arguments. For example, `sum(1,3,5)` is equivalent to `sum(vector(1,3,5))`.

2.12.5 **Computing correlations – cor()** If `x` is a data matrix, `cor(x)` computes the correlation matrix of `x`. If `x` is *n* by *m* , `cor(x)` is *m* by *m* , with 1's down the diagonal and `cor(x)[i,j]` the Pearson correlation between `x[,i]` and `x[,j]`.

```
Cmd> w <- matrix(vector(45.5,42.1,53.8,48.5,44.5,\
58.4,74.1,72.0,63.8,67.7,  28.7,35.9,32.1,28.5,28.1),5)

Cmd> w # 3 columns
(1,1)         45.5         58.4         28.7
(2,1)         42.1         74.1         35.9
(3,1)         53.8           72         32.1
(4,1)         48.5         63.8         28.5
(5,1)         44.5         67.7         28.1

Cmd> cor(w) # 3 by 3 matrix
(1,1)            1     0.049932     -0.16287
(2,1)     0.049932            1      0.78611
(3,1)     -0.16287      0.78611            1
```

If `x1`, `x2`, ... are vectors or matrices all with the same number of rows, then `cor(x1,x2,x3,...)` is equivalent to `cor(hconcat(x1,x2,x3,...))` (see Sec. 2.10.6).

```
Cmd> makecols(w,w1,w2,w3) # See Sec. 2.11.3;

Cmd> cor(w1,w2,w3) # same as cor(w)
(1,1)            1     0.049932     -0.16287
(2,1)     0.049932            1      0.78611
(3,1)     -0.16287      0.78611            1
```

You can use subscripts (Sec. 2.8.14) to get the correlation between two variables as a single number:

```
Cmd> cor(w1,w2)[1,2] # compute a single correlation coef
(1,1)     0.049932
```

If there are any `MISSING` values in the arguments, correlations are computed using only complete cases, effectively deleting any row with any `MISSING` values.

```
Cmd> ww <- w; ww[2,3] <- ? # put a MISSING value in row 2
```

```
Cmd> cor(ww) # same as cor(w[-2,])
WARNING: 1 cases with missing values deleted in cor
(1,1)           1      0.64695      0.92817
(2,1)     0.64695            1      0.65979
(3,1)     0.92817      0.65979            1
```

2.12.6 **Student's *t* related functions** There are four functions for statistical analyses based on Student's *t*. Functions `tval()` and `t2val()` calculate one- and two-sample *t*-statistics and `tint()` and `t2int()` compute one- and two-sample confidence intervals.

`tval(x)`, where `x` is a REAL vector, computes the usual one-sample *t*-statistic to test the null hypothesis $H_0$: $\mu_x = 0$. You can test a different null hypothesis, say $H_0$: $\mu_x = 100$ by the statistic computed by `tval(x-100)`. When `x` and `y` are REAL vectors of the same length, you can compute the *paired t*-statistic to test the null hypothesis $H_0$:$\mu_x = \mu_y$ by `tval(x-y)`.

```
Cmd> x1 <- vector(3,2,5,4,6,8,6,4,3,7,3)

Cmd> vector(tval(x1), tval(x1 - 5)) # test H0: μ = 0 and H0 μ = 5
(1)       8.0437     -0.63088  1-sample t-statistics

Cmd> x2 <- vector(4,2,5,6,7,9,7,4,3,8,5); tval(x2-x1)
(1)       3.6145              Paired t-statistic
```

When `x` and `y` are REAL vectors, `t2val(x,y)` computes a two-sample *t*-statistic to test the null hypothesis $H_0$: $\mu_x = \mu_y$ or, equivalently $H_0$: $\mu_x - \mu_y = 0$. To test a different null hypothesis, say $H_0$: $\mu_x - \mu_y = 10$, use `t2val(x-10,y)`. This use of `t2val()` assumes the samples come from populations with equal variances and uses a pooled estimate of the standard deviation. See below for the use of `t2val()` when you cannot assume equal variances.

```
Cmd> y <- vector(7,9,6,5,7,6,9,8)

Cmd> # test H0:μx = μy and H0:μx-μy = -3

Cmd> vector(t2val(x1,y), t2val(x1-(-3),y))
(1)      -3.0795      0.63278  2-sample t-statistics
```

See Sec. 2.12.7 to see how to compute P-values for these test statistics.

Functions `tint()` and `t2int()` compute one- or two- sample confidence intervals based on Student's *t*. In a sense, they are complementary to `tval()` and `t2val()`. Their output is a REAL vector of length 2 containing the lower and upper ends of the confidence interval.

When `x` is a REAL vector and `coverage` is a REAL scalar, `tint(x,coverage)` returns a one-sample confidence interval for the mean $\mu_x$ with confidence level `coverage`. For obvious reasons, 0 < `coverage` < 1. For instance, you can compute a 95% confidence interval by `tint(x,.95)`.

When `x` and `y` are two REAL vectors of data, `t2int(x,y,coverage)` computes a two-sample confidence interval for the difference $\mu_x - \mu_y$ of their means, where `coverage` is as for `tint()`. Like `t2val()`, function `t2int()` assumes equal variances and uses a

pooled estimate of the common variance in computing the standard error of the difference of means. See below for usage not requiring equal variances.

```
Cmd> tint(x1,.95)
(1)       3.3521       5.9207  95% confidence interval for μ
Cmd> t2int(x1,y,.95)
(1)      -4.1936     -0.78364  95% confidence interval for μx-μy
```

Both `t2val()` and `t2int()` assume by default that $\sigma_1^2 = \sigma_2^2 = \sigma^2$, where $\sigma_1^2$ and $\sigma_2^2$ are variances of the two populations being compared. Because of this assumption they use the estimated standard error of $\bar{x}_1 - \bar{x}_2$, $s_{\bar{x}_1 - \bar{x}_2} = s_p\sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$, where $s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2}$ is the "pooled" estimate of $\sigma^2$ based on the two sample variances $s_1^2$ and $s_2^2$ and has $n_1 - n_2 - 2$ degrees of freedom. When the variances cannot be assumed equal, you can use the "unpooled" standard error $s_{\bar{x}_1 - \bar{x}_2} = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}$. Unfortunately Student's *t*-distribution is no longer directly applicable. However, a good approximation is to use Student's *t* with estimated degrees of freedom $f = \frac{(v_1 + v_2)^2}{\frac{v_1^2}{n_1 - 1} + \frac{v_2^2}{n_2 - 1}}$ where $v_i = s_i^2 / n_i$, $i$ = 1, 2 (Snedecor and Cochran 1980, Sec. 6.11). Including `pooled:F` as an extra argument to `t2val()` and `t2int()` directs MacAnova to use this approximation. For `t2val()` the result returned is a structure with components `t` and `df`.

```
Cmd> t2int(x1,y,.95,pooled:F) # slightly different from before
(1)      -4.1205     -0.85676

Cmd> t2val(x1,y,pooled:F) # not assuming same variances
component: t
(1)      -3.2186       Value of t-statistic
component: df
(1)       16.926       Estimated degrees of freedom
```

2.12.7 **P-values and cumulative distribution functions** MacAnova has several functions for computing the cumulative distribution functions (CDF) of standard probability distributions. You can use them to compute P-values associated with many standard test statistics.

The arguments to these functions are always a value `x` of the random variable of interest, followed by the value of parameters as indicated in the following table:

| Distribution | Usage |
|---|---|
| Standard normal | `cumnor(x)` |
| Student's *t* on `df` degrees of freedom | `cumstu(x,df)` |
| Non-central Student's *t* on `df` degrees of freedom, noncentrality `delta` | `cumstu(x,df,delta)` |
| $\chi^2$ on `df` degrees of freedom | `cumchi(x,df)` |
| Non-central $\chi^2$ on `df` degrees of freedom, noncentrality `noncen` | `cumchi (x,df,noncen)` |
| *F* on `df1` and `df2` degrees of freedom | `cumF(x,df1,df2)` |
| Non-central *F* on `df1` and `df2` degrees of freedom, noncentrality `noncen` | `cumF(x,df1,df2,noncen)` |
| Gamma with shape parameter `alpha` | `cumgamma(x,alpha)` |
| Beta with parameters `alpha` and `gamma` | `cumbeta(x,alpha,gamma)` |
| Non-central beta with parameters `alpha` and `gamma`, noncentrality `noncen` | `cumbeta(x,alpha,gamma,noncen)` |
| Binomial with `n` trials and success probability `p` | `cumbin(x,n,p)` |
| Poisson with mean `lambda` | `cumpoi(x,lambda)` |
| Studentized range for `k` groups, `df` error degrees of freedom | `cumstudrng(x,k,df)` |

The noncentral versions of `cumchi()`, `cumstu()`, `cumF()` and `cumbeta()` are useful in computing the powers of tests, that is, the rejection probability when the null hypothesis is not true.

In every case, these CDF functions compute a *lower tail* probability, that is, the probability that the random variable is less than or equal to `x`.

All the parameters except noncentrality parameters must be positive. Parameter `noncen` must be non-negative for `cumchi()`, `cumF()` and `cumbeta()`. Parameter `n` for `cumbin()` and `k` for `cumstudrng()` must be integers, but other parameters, including degrees of freedom, need not be integers. For `cumbin()`, `n` must be a positive integer and `p` must be between 0 and 1.

For all functions, when `x` is a vector or matrix, the probability is computed for each element of `x`, producing a vector or matrix. Similarly the parameter arguments may be vectors or matrices. The sizes and shapes of the arguments must match, except that `x` or any of the parameters may always be a scalar.

These functions can be used to compute P-values based on the observed value $x_{obs}$ of a test statistic $X$. The lower tail probability $P(X \le x_{obs})$ that each function computes can be directly interpreted as a *P*-value for a one-tail test that rejects for values in the left tail of the test statistic. Because many standard tests such as $\chi^2$ and *F* are *upper tail* tests that reject the null hypothesis only for large positive values, you often need an upper tail P-value $P(X \ge x_{obs}) = 1 - P(X < x_{obs})$. You get this by subtracting the result

from 1. For example, `1 - cumchi(15.3,10)` computes the P-value for a $\chi^2$ statistic on 10 degrees of freedom with observed value 15.3. Of course, for the binomial and Poisson distributions which are discrete, when $x_{obs}$ is an integer , $P(X \geq x_{obs}) = 1 - P(X \leq x_{obs} - 1)$.

To compute a two-tail P-value for a *z*- or *t*-test, use `2*(1-cumnor(abs(xobs)))` or `2*(1-cumstu(abs(xobs),df))`. You can compute a two-tail P-value for an F-test of equal variance based on estimated variances `var1` and `var2` as

```
2*min(cumF(var1/var2,df1,df2), 1-cumF(var1/var2,df1,df2)).
```

```
Cmd> vector(cumnor(1.96), cumstu(-2.1,10)) # normal, Student's t(10)
(1)        0.975     0.031039

Cmd> 2*(1 - cumstu(abs(-2.1),10)) # 2-tail t P-value
(1)     0.062077

Cmd> vector(1-cumchi(34,25),1-cumF(4.5,2,10)) # chi-squared and F
(1)      0.10791     0.040386

Cmd> 1-cumstu(2.1,vector(10,20,30,40,50)) # 5 d.f.s at once
(1)     0.031039     0.024309     0.022121     0.021041     0.020398

Cmd> xbar <- vector(150.91,150.77,159.51,156.43,165.54)#enter means

Cmd> spooled <- 15.974 # enter pooled var from k=5 groups of n=20

Cmd> n <- 20; k <- 5

Cmd> studrng <- (max(xbar) - min(xbar))/(spooled/sqrt(n));studrng
(1)       4.1351   Studentized range statistic

Cmd> 1-cumstudrng(studrng, k, k*(n-1)) # compute P-value
(1)     0.034262   Significant at 5% level
```

You can compute binomial and Poisson probabilities of the form P(X = x) by taking differences of cumulative probabilities.

```
Cmd> p<-.3;n <-9; p8 <- cumbin(8,n,p) - cumbin(7,n,p); p8
(1)   0.00041334   P(X=8)

Cmd> (n/1)*p^8*(1-p)^(n-8) # (n/1) is binomial coefficient (n  1)
(1)   0.00041334   Confirmation of value
```

You can even compute P(X=0),..., P(X=n) in a single line.

```
Cmd> cdf <- cumbin(run(0,n),n,p); cdf - vector(0,cdf[-(n+1)])
 (1)     0.040354      0.15565      0.26683      0.26683      0.17153
 (6)     0.073514     0.021004    0.0038579   0.00041334   1.9683e-05
```

The studentized range is $R/s$, where $R = x_{max} - x_{min}$ is the range from a normal sample of size `k` with standard deviation $\sigma$, and $s$ is an independent estimate of $\sigma$ with `df` degrees of freedom. It is often applied with data from $k$ independent samples each of size $n$, with $R/s, = \{(\bar{x})_{max} - (\bar{x})_{min}\} / \{s_{pooled} / \sqrt{n}\}$, where $s^2_{pooled}$ is the error mean square from an analysis of variance.

Non-central *t* is the ratio $\frac{z+\delta}{\sqrt{\frac{1}{f}\chi^2_f}}$, where *z* is standard normal, independent of $\chi^2_f$ in the denominator. Non-central $\chi^2_f(\lambda)$ with non-centrality $\lambda$ is $\sum_{i=1}^{f}(z_i+\delta_i)^2$, where $z_1$, $z_2$, ..., $z_f$ are independent standard normal and $\lambda = \sum_{i=1}^{f}\delta_i^2$. Non-central $F_{f,g}(\lambda) = \frac{\chi^2_f(\lambda)/f}{\chi^2_g/g}$, where the numerator and denominator are independent. Non-central $\beta_{\alpha,\gamma}(\lambda) = \frac{\chi^2_f(\lambda)}{\chi^2_f(\lambda)+\chi^2_g}$, $f = 2\alpha$, $g = 2\gamma$, where the two $\chi^2$'s are independent.

2.12.8 **Probability points and inverse cumulatives** MacAnova can compute probability points or critical values (inverse cumulatives) for many of the distributions in Sec. 2.12.7. Most require you to specify degrees of freedom or shape parameters. The arguments to these functions are always the probability value of interest followed by distribution parameters as indicated in the following table:

| Distribution | Usage |
|---|---|
| Standard normal | `invnor(p)` |
| Student's *t* on `df` degrees of freedom | `invstu(p,df)` |
| $\chi^2$ on `df` degrees of freedom | `invchi(p,df)` |
| Non-central $\chi^2$ on `df` degrees of freedom and non-centrality parameter `lambda` | `invchi(p,df,lambda)` |
| *F* on `df1` and `df2` degrees of freedom | `invF(p,df1,df2)` |
| Gamma with shape parameter `alpha` | `invgamma(p,alpha)` |
| Beta with parameters `alpha` and `gamma` | `invbeta(p,alpha,beta)` |
| Studentized range for `k` groups, `df` error degrees of freedom | `invstudrng(p,k,df)` |

In every case, these inverse CDF functions compute a value $x_0$ such that $P(x \le x_0) = p$. That is, they are the inverses of the corresponding CDF functions in Sec. 2.12.7.

The value of `p` must always be between 0 and 1. All the other parameters except `k` for `invstudrng()`, including degrees of freedom, must be positive but need not be integers.

When `p` is a vector or matrix, the inverse CDF is computed for each element of `p` producing a vector or matrix. Similarly the parameters may be vectors or matrices. The sizes and shapes of the arguments must match, except that `p` or any of the parameters may always be scalars.

```
Cmd> invnor(vector(.10,.05,.025,.01,.005)) # normal probability
points
(1)      -1.2816      -1.6449       -1.96      -2.3263      -2.5758
```

```
Cmd> invstu(.975,run(5,25,5)) # Student's t on 5, 10, 15, 20, 25 df
(1)       2.5706       2.2281       2.1314        2.086       2.0595

Cmd> invF(.95,5,run(5,25,5)) # F with df1=5, and df2=5,10,15,20,25
(1)       5.0503       3.3258       2.9013       2.7109        2.603

Cmd> invstudrng(1-vector(.1, .05, .01, .001), k, k*(n-1))
(1)       3.5309       3.9328       4.7373       5.7199
```

The `invstudrng()` output are the 10%, 5%, 1% and 0.1% critical values of the Studentized range statistic based on 5 samples of size 20.

You can actually compute a page of an F-table with a single command. The following computes the upper 5% point of F for 1 through 8 numerator degrees of freedom and 1 through 5 denominator degrees of freedom.

```
Cmd> invF(1-.05,run(8)*rep(1,5)',rep(1,8)*run(5)')
(1,1)      161.45       18.513       10.128       7.7086       6.6079
(2,1)       199.5           19       9.5521       6.9443       5.7861
(3,1)      215.71       19.164       9.2766       6.5914       5.4095
(4,1)      224.58       19.247       9.1172       6.3882       5.1922
(5,1)      230.16       19.296       9.0135       6.2561       5.0503
(6,1)      233.99        19.33       8.9406       6.1631       4.9503
(7,1)      236.77       19.353       8.8867       6.0942       4.8759
(8,1)      238.88       19.371       8.8452        6.041       4.8183
```

2.12.9 **Grouping data in class intervals – bin()** An important method of summarizing a data set is to compute a grouped frequency distribution by counting the number of values that lie in a set of class intervals defined by class boundaries $b_1 \le b_2 \le b_3 \le \ldots \le b_k \le b_{k+1}$. Class interval $j$ consists of all values between $b_j$ and $b_{j+1}$. Depending on who is making the definition, either the left end $b_j$ or the right end $b_{j+1}$ of the interval is considered in the interval. The counts in each interval are the *frequencies*. Function `bin()` allows you to make this summary. It is probably best explained by example.

```
Cmd> x <- vector(33,35, 41,41,42,44,47,49,49,50,50,50,\
51,53,53,54,54,54,55,57,58,60 ,61,62,67) # enter sorted data

Cmd> b <- vector(30,40,50,60,70)# or run(30,70,10); class limits

Cmd> bin(x,b) # group x using boundaries b
component: boundaries
(1)          30           40           50           60           70
component: counts
(1)           2           10           10            3
```

The first argument, `x`, contains the data to be grouped; the second, `b`, is a vector of class limits. Here `b` defines the boundaries of 4 = `length(b)-1` classes or groups. Group 1 consists of all values between `b[1]` and `b[2]`, group 2 consists of all values between `b[2]` and `b[3]`, and so on. The data are entered in sorted order only to make it easier to check that `bin()` is doing what it should.

The result is a structure with components `boundaries` and `counts`. Component `boundaries` is identical with `b`, while `counts[j]` is the number of input values in

class `j`, that is, the frequency of the class. Here there are two values (33 and 35) in class 1, ten values (41, 41, 42, 44, 47, 49, 49, 50, 50, 50) in class 2, and so on. By default, any value exactly on a class limit at the *right* or upper end of a class is counted in that class so, for example, the three 50's are considered to be in class 2 together with 41 through 49, and 60 goes in class 3.

Any value not in any class (here ≤30 or > 70) is not counted, but a warning message is printed. You can suppress the warning message by including `silent:T`) as an argument.

```
Cmd> bin(vector(x,25,75),b) # data vector enlarged by 2 values
WARNING: 1 low and 1 high values not counted by bin()
component: boundaries
(1)           30           40           50           60           70
component: counts
(1)            2           10           10            3
```

Some statistics books recommend that you include the *left* or lower end of a class interval in the interval instead of the right end. You can follow this convention by including `leftendin:T` as an argument.

```
Cmd> bin(x,b,leftendin:T) # lower end of interval in class
component: boundaries
(1)           30           40           50           60           70
component: counts
(1)            2            7           12            4
```

Now the new frequencies reflect the fact that the three 50's are put in class 3 and the 60 in class 4.

When you want equally spaced boundaries, the usage, `bin(x,vector(b0,width))`, can simplify things. `b0` is interpreted as a "typical" class limit and `width` is the desired class width. Enough boundaries of the form `b` ± *j*×`width` are computed so that *all* the data are included.

```
Cmd> bin(x,vector(30,10)) # gives the same result as bin(x,b)
component: boundaries
(1)           30           40           50           60           70
component: counts
(1)            2           10           10            3
```

Instead of `vector(30,10)`, you could use `vector(0,10)` or even `vector(-100,10)` and still get the same output.

Still easier and OK for quick summaries is `bin(x,nclasses)`, where `nclasses` is the number of classes wanted. `bin()` chooses `nclasses`+1 equally spaced class limits which include all the data. No attempt is made to ensure the values are "neat."

```
Cmd> bin(x,4); bin() chooses boundaries
component: boundaries
(1)        32.15         41.5        50.85         60.2        69.55
component: counts
(1)            4            8           10            3
```

Even easier is simply `bin(x)` for which `bin()` selects both the number of classes and the width.

```
Cmd> bin(x)
component: boundaries
(1)        32.15       38.383       44.617        50.85       57.083
(6)       63.317        69.55
component: counts
(1)            2            4            6            8            4
(6)            1
```

You can use keyword phrase `leftendin:T` with all these usages.

If `x` is a `REAL` matrix, rather than a vector, each column is grouped, all with the same class limits. Component `boundaries` is as before, but now component `counts` is a matrix, with each column containing the frequencies for the data in the corresponding column of `x`.

2.13 **Random numbers – runi() and rnorm()** MacAnova has functions for generating standard normal and uniformly distributed pseudo-random numbers.

`runi(n)` returns a vector of `n` random variables that are uniformly distributed between 0 and 1. `n` must be a positive integer.

`rnorm(n)` returns a vector of `n` random variables which have standard normal ($\mu = 0$, $\sigma = 1$) distribution.

2.13.1 "**Seeding" runi() and rnorm() – setseeds() and getseeds()** The values computed by `runi()` and `rnorm()` depend on two internal positive integer "seeds" which are updated every time a number is computed. If the seeds are the same on two different occasions you will get exactly the same sequence of random numbers.

You can set the seeds by `setseeds(vector(seed1,seed2))` or `setseeds(seed1, seed2)`, where `seed1` and `seed2` are positive integers between 1 and 2147483399. `setseeds(0,0)` initializes the seeds with values based on the date and time providing a more or less random starting point for random numbers. If you don't explicitly set the seeds, the first time random numbers are computed the seeds are initialized as if you had typed `setseeds(0,0)`.

You can retrieve current values of the seeds in a vector of length 2 by function `getseeds()`. If `getseeds()` is `vector(0,0)`, the seeds have not been set. Resetting the seeds to values previously retrieved makes it possible to generate the same sequence of random numbers more than once.

Here are some simple examples.

```
Cmd> runi(5) # 5 uniforms without initializing
WARNING: starting random number seeds are 1979189978 and 1730035780
(1)       0.2765     0.046009      0.27089      0.21016      0.16606

Cmd> rnorm(5) # 5 normals
(1)       1.9504     -0.23464     -0.58285      0.43007     -0.09322

Cmd> setseeds(1979189978,1730035780) #reset seeds,

Cmd> runi(5) #same vals
(1)       0.2765     0.046009      0.27089      0.21016      0.16606
```

```
Cmd> getseeds() # value is "invisible"; output is "side effect"
Seeds are 1849669167 and 1493059652
```

By using `runi(n)` as argument `p` in the inverse cumulative distributions functions described in Sec. 2.12.8, you can easily generate pseudo-random samples of Student's *t*, *F*, $\chi^2$, gamma, and beta distributed random variables.

```
Cmd> f<-invF(runi(5),10,30); f # small random sample from F(10,30)
(1)      0.72673       1.6098       2.4452      0.46853      0.76309
```

2.14 **More on rep() and run()** Functions `run()` and `rep()` are very useful for setting up factor variables in ANOVA models (described in Chapter 3). As described in Sec. 2.8.12, `run()` generates equally spaced sequences and `rep()` replicates its first argument. We saw there that `run(n)` returned `vector(1,2,...,n)` and `rep(v,n)` returned `vector(v,v,...,v)`, where there are `n` repetitions and where `v` is a scalar or vector.

```
Cmd> rep(run(4),3) # 3 replicates of 1,2,3,4
 (1)           1           2           3           4           1
 (6)           2           3           4           1           2
(11)           3           4
```

There is a more general usage of `rep()`. In `rep(v,counts)`, where `counts` is a vector of non-negative integers *of the same length* as `v`, the corresponding `v[i]` is repeated `counts[i]` times. In particular, `rep(v,rep(m,n))`, where `n` is the length of `v`, produces `m` copies of each element of `v`.

```
Cmd> rep(run(3),rep(4,3)) # 4 copies of each element of run(3)
 (1)           1           1           1           1           2
 (6)           2           2           2           3           3
(11)           3           3
```

Thus you might use `rep(run(4),3)` and `rep(run(3),rep(4,3))` to generate treatment number and block numbers associated with a randomized block design with 4 treatments and 3 replicates.

```
Cmd> rep(run(4),vector(2,1,0,6)) # 2 1's, 1 2, no 3's and 6 4's.
(1)           1           1           2           4           4
(6)           4           4           4           4
```

2.15 **Making graphs** MacAnova has several functions and macros for making graphs of data. The basic functions are as follows:

| Function or Macro | Description |
|---|---|
| `plot(x,y)` | Scatter plot or impulse plot of `y` versus `x`, points optionally connected by lines |
| `lineplot(x,y)` | Connected line plot of x versus y, optionally augmented by "impulses" |
| `chplot(x,y,sym)` | Scatter plot or impulse plot of `y` versus `x`, using user supplied symbols, points optionally connected by lines |
| `showplot()` | Redisplay previously plotted graph, possibly with changed labels or plotting limits |
| `colplot(x)` | Makes parallel line plots of matrix `x` against row number |
| `rowplot(x)` | Makes parallel line plots of matrix `x` against column number |

"Impulses" are vertical lines connecting a point with the x-axis.

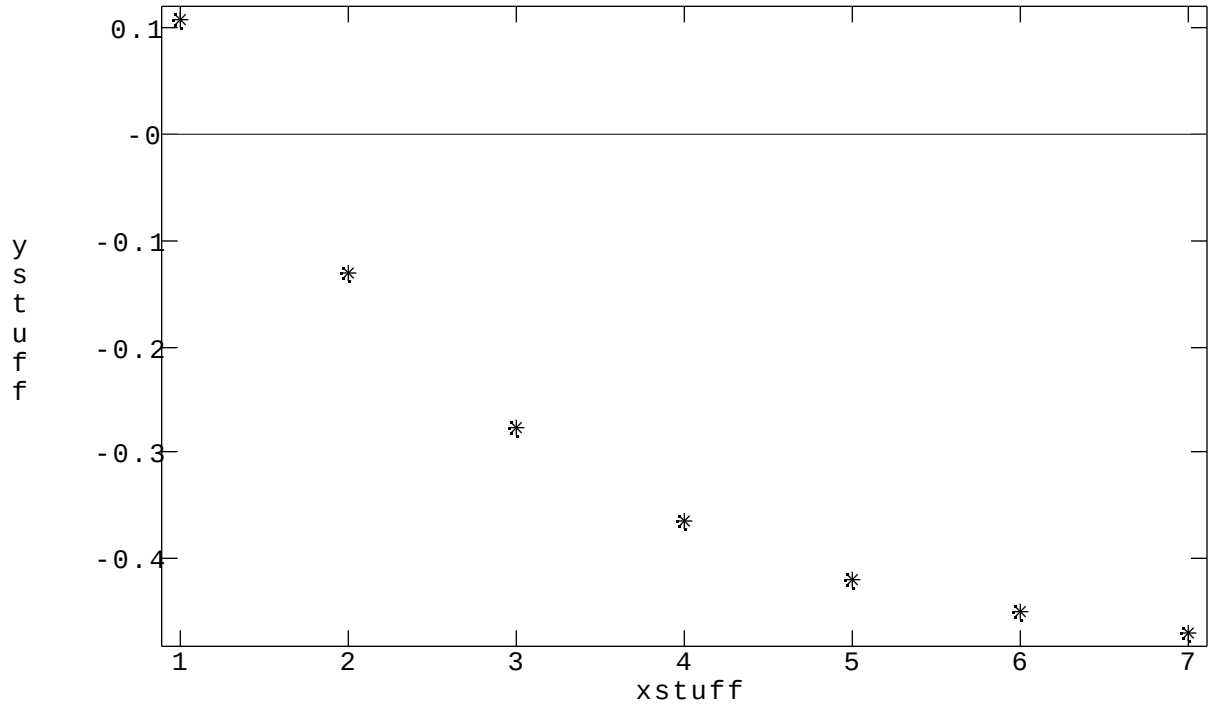
`plot()`, `lineplot()` and `chplot()` take at least two arguments `x` and `y`, where `x` is a REAL vector and `y` is a REAL vector or matrix, and plot each column of `y` against the values of `x`. The length of `x` must normally match the number of rows in `y`, but see below for a useful exception.

All plotting commands also recognize several optional keyword arguments which you can use to specify a title, axis labels, and minimum and maximum values to be plotted.

2.15.1 **plot()** `plot(x,y)` does a simple scatter plot of `y` versus `x`. If `y` has several columns, a different symbol is used for each column.

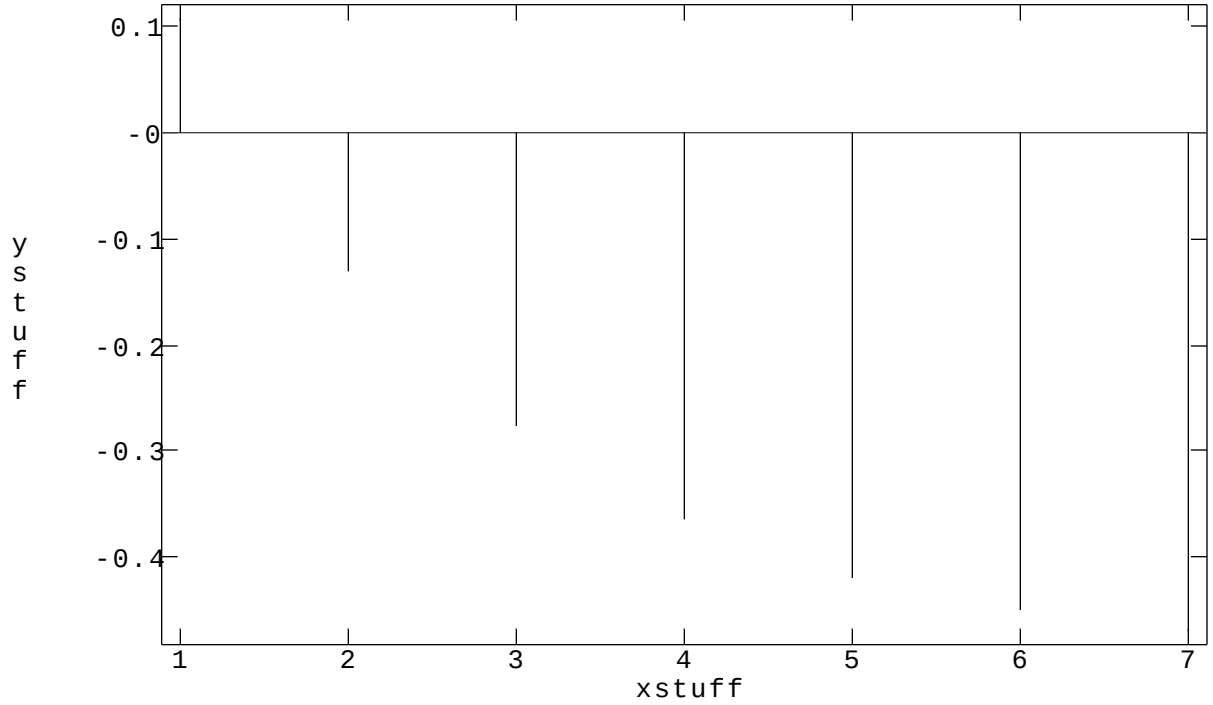
```
Cmd> xstuff<-vector(1,2,3,4,5,6,7); ystuff <-exp(-.5*xstuff)-.5
```

```
Cmd> plot(xstuff,ystuff) # simplest usage
```



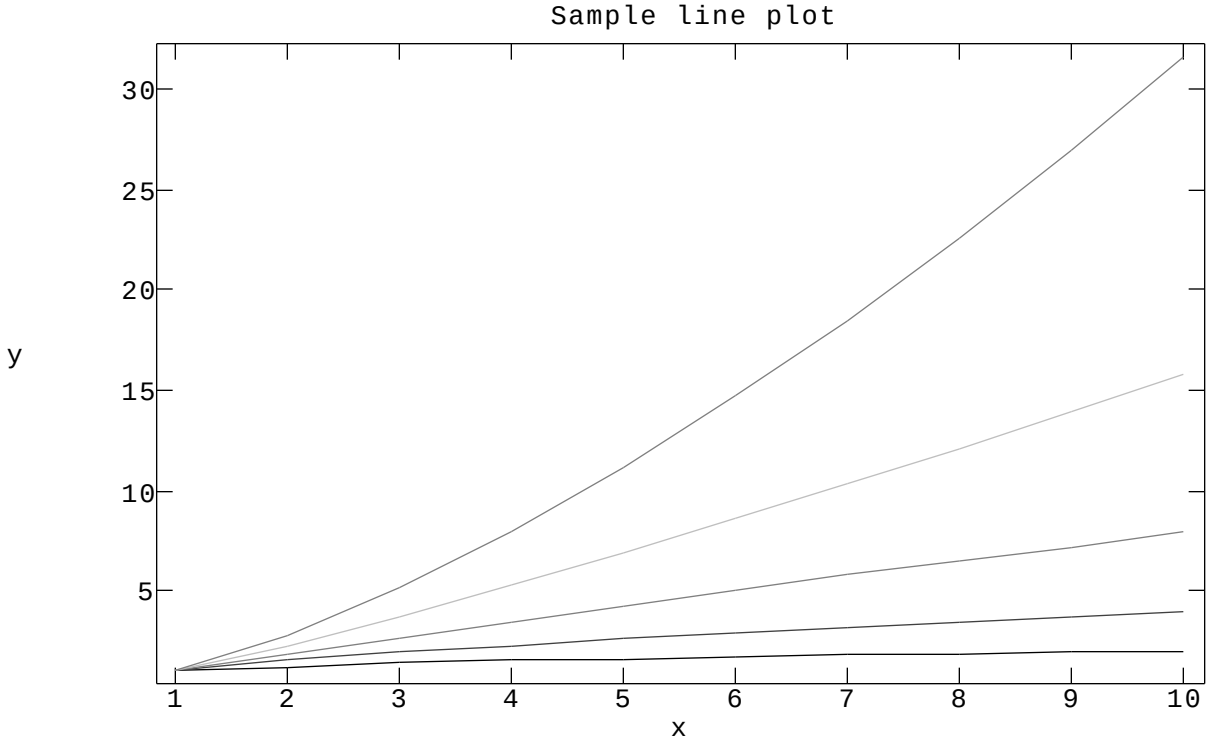

Note that the axes are labelled with the variable names.

```
Cmd> plot(xstuff,ystuff,impulse:T) # make impulse plot
```



2.15.1 **lineplot()** `lineplot(x,y)` draws lines between the successive points plotted, but does not draw any symbol at the points. Normally, the elements of `x` should be in monotonic (increasing or decreasing) order. `lineplot()` uses different line types (solid, dashed, dotted, ...) for successive columns of `y`. You can use keyword `linetype` to change the default line types. See Sec. 8.5.1.

```
Cmd> lineplot(x:run(10),y:run(10)^(.3*run(5)'),\
title:"Sample line plot") #Note keywords x and y to label axes
```




2.15.1 **chplot()** `chplot(x,y,sym)`, where `sym` is a `REAL` or `CHARACTER` vector or matrix specifying symbols or characters to be plotted, is similar to `plot(x,y)`. When `sym` is a matrix, it must have the same number of columns as `y`.

When `sym` is `REAL`, its elements must be integers between 1 and 999 and the numbers themselves are used as the plotting symbols.

When `sym` is a `CHARACTER` vector or matrix, its elements are used as plotting symbols (actually only the first three characters are used if there are more). For example, `chplot(run(5),run(5)^2, vector("Cat","Dog","Wolf","Sheep","Bird"))` labels the successive points with `Cat`, `Dog`, `Wol`, `She` and `Bir`.

Character symbols `"\1"`, `"\2"`, `"\3"`, `"\4"`, `"\5"`, `"\6"`, `"\7"`, `"\10"` are special and are interpreted as diamond, plus sign, cross, triangle, star, dot and small cross. For example, if `y` is a vector, `chplot(x,y,"\6")` is identical to `plot(x,y)`.

When `sym` has a single element as in `chplot(x,y,"#")`, it is used for all points plotted.

When `sym` is a vector whose length matches the number of columns in `y`, then `sym[j]` labels all points associated with *column* `j`, that is the point `y[i,j]` is plotted against

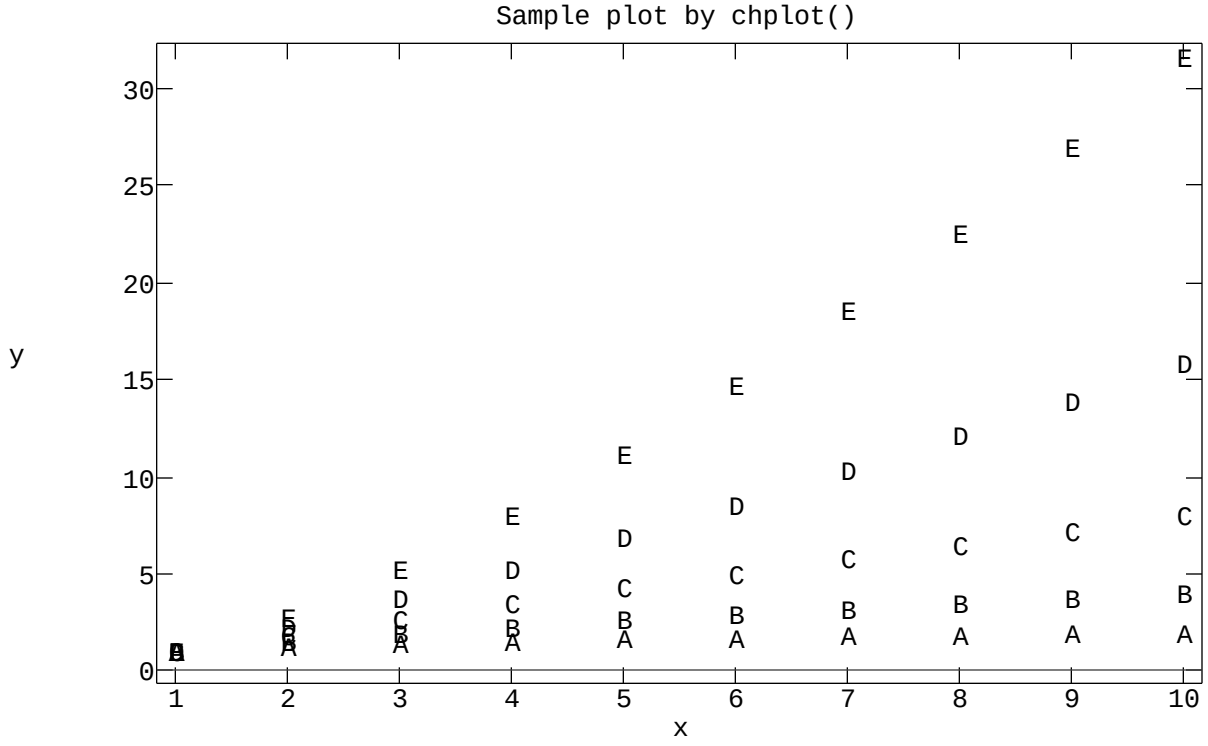
`x[i]` with character or number `sym[j]`. For example, if `y` has 3 columns, `chplot(x,y,run(3))`, labels all the points from `y[,1]`, `y[,2]` and `y[,3]` with `"1"`, `"2"` and `"3"`, respectively.

When the `sym` is a vector whose length does *not* match the number of columns in `y`, then `sym[j]` labels *row* `j`, with the symbols *repeating cyclically* if `length(sym) < nrows(y)`.

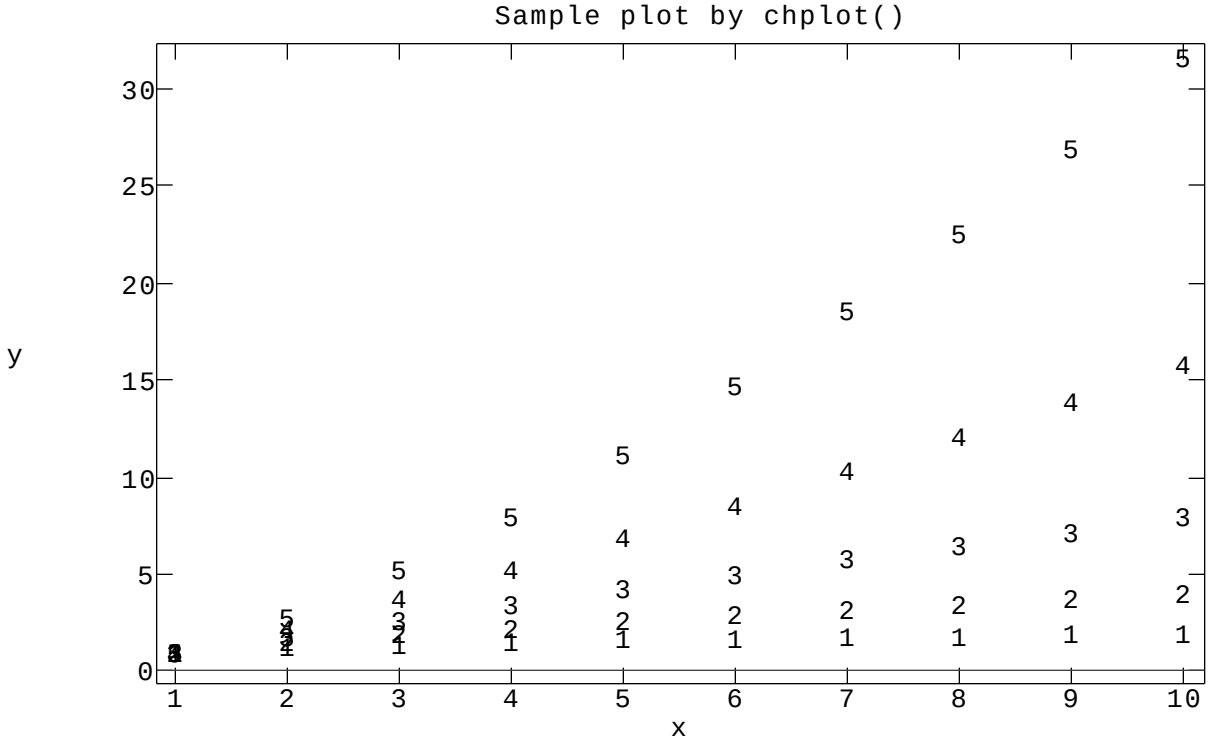
Finally, when `sym` is a matrix it must have the same number of columns as `y`, and symbols for plotting the values from row *i* of `y` are taken from row *i* of `sym`, repeating rows of `sym` cyclically if necessary.

You can omit `sym` altogether; if `y` has only one column, each point is labelled with the row number; if `y` has several columns, all the points from a column are labelled with the column number.

```
Cmd> chplot(x:run(10),y:run(10)^(.3*run(5)'),\
vector("A","B","C","D","E"),ymin:0,\
title:"Sample plot by chplot()")
```



```
Cmd> chplot(x:run(10),y:run(10)^(.3*run(5)'),ymin:0,\
title:"Sample plot by chplot()") # no plotting symbols provided
```




2.15.4 **Equally spaced x values** The exception to the requirement that `x` and `y` have the same number of rows is a convention you can use when you want to plot the rows of `y` against equally spaced values on the X-axis. When `x` is a `REAL` scalar (single number), `y[i,]` is plotted against `vector(x,x+1,x+2,...)`, that is they are equally spaced by 1 starting with `x`. When `x` is a vector of length 2, say, `vector(x0,d)`, `y[i,]` is plotted against vector(`x0,x0+d, x0+2*d,...`), that is equally spaced starting with `x0` and incrementing by `d`. For example, `plot(1,y)` is equivalent to `plot(run(nrows(y)),y)` and `lineplot(vector(1967,1/12),y)` might be used to plot monthly values for which `y[1,]` was data for January, 1967.

2.15.5 **Graphics keywords** See Sec. 8.5 for information on the use of keywords `xmin`, `ymin`, `xmax`, `ymax`, `xlab`, `ylab`, `xaxis`, `yaxis`, `title`, `file`, `keep`, and `show`, and on how to add information to previously created graphs.

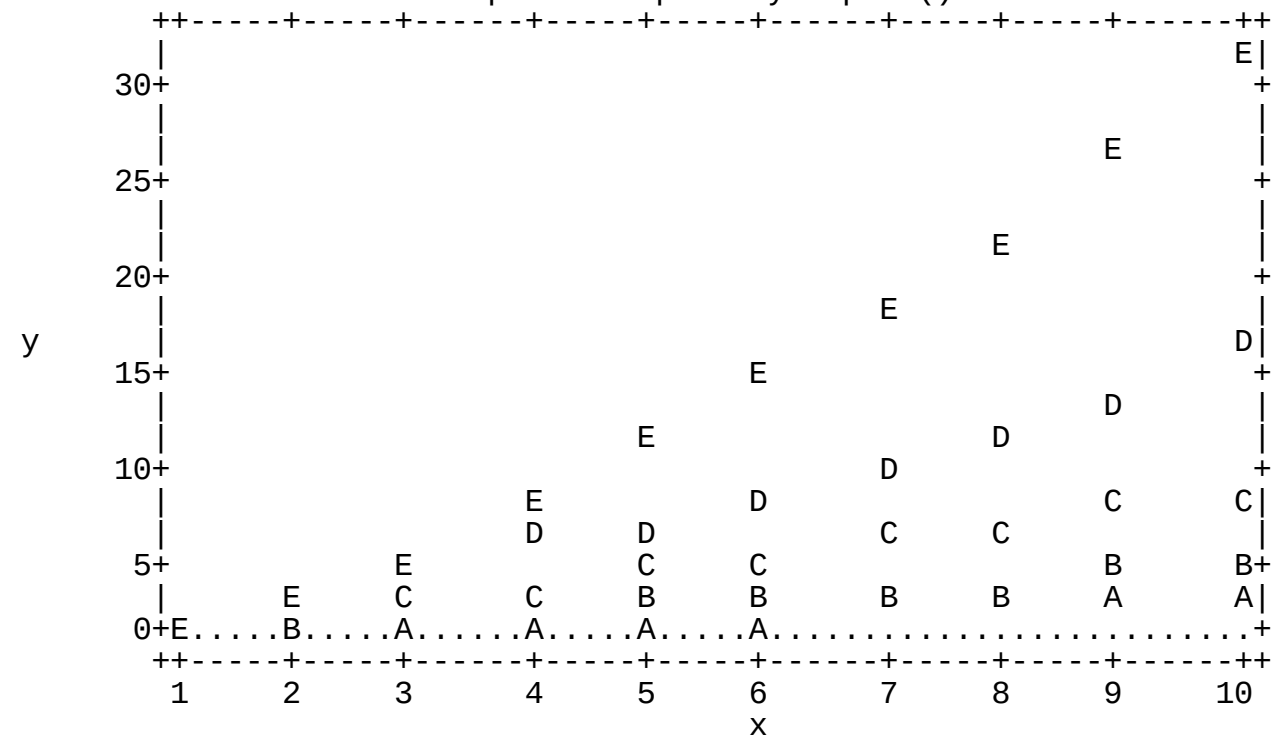
You can draw lines between points plotted by `plot()` or `chplot()` by using keyword phrase `lines:T`.

On all three commands, you can use `impulses:T` add "impulses" to the plot. On `plot()` but not on `chplot()`, this suppresses the plotting symbols.

2.15.6 **colplot and rowplot** These are pre-defined macros that use `chplot()` to draw "interaction plots" of data in a matrix `x`. `colplot(x)` draws line plots of each *column* of `x` versus the row numbers `run(nrows(x))`. Data points are labelled with the column number. Similarly `rowplot(x)` draws line plots of each *row* of `x` versus the column numbers `run(ncols(x))`. See Sec. 10.12 for an example of the use of `colplot`.

You can use keyword phrase `dumb:T` as an argument with any plotting command, as in `plot(x,y,dumb:T)`. This results in a crude plot made up of printable characters being printed. For many purposes this is sufficient to get a sufficiently clear picture of the relationships being plotted, and has the advantage that it is printable on any printer and is written along with other output to a spool file (see. Sec. 2.16).

```
Cmd> chplot(x:run(10),y:run(10)^(.3*run(5)'),\
vector("A","B","C","D","E"),ymin:0,\
title:"Sample dumb plot by chplot()",dumb:T)
```




In the Macintosh and Windows versions of MacAnova, you can copy graphs to the Clipboard and then paste them into word processor or graphic editing documents, and you can print them using **Print Graph** on the **File** menu (see Appendices B and D).

2.16 **Using spool() to save output** You often want to keep track of exactly what you have done and what were the results. Command `spool()` allows you to "spool" your session, that is, save in a file a transcript of what you typed and what MacAnova printed. Spooling is a little bit like using a tape recorder – everything, wanted or unwanted is recorded. After leaving MacAnova, you can edit the file into a report using a word processor, or simply print it as is.

`spool(fileName)`, where `fileName` is a quoted string or `CHARACTER` variable containing a legal file name, starts spooling on the named file. From that point on until you stop or suspend spooling, everything you type and everything MacAnova responds (except high resolution graphs) is written to the file in plain text (ASCII) form. In a windowed version (Macintosh, Windows or Motif), if `fileName` is the empty name `""`, you will be able to specify the file using a dialog box. On a Macintosh, selecting **Spool Output To File** on the **File** menu is another way to start spooling.

If the spool file already exists, `spool()` normally starts writing at its end. This allows you to accumulate output from several runs in a single file. If this is not what you want, use `spool(fileName,new:T)` which starts transcription at the start of the file, destroying any information already there.

To suspend spooling, simply type `spool()`, with no argument. On a Macintosh you can select **Stop Spooling** on the **File** menu. If spooling has been suspended, `spool()`, with no argument restarts it. On a Macintosh you can select **Resume Spooling** on the **File** menu.

For example, you might suspend spooling while you experiment with different analyses of data, and then resume it once you have a better idea of what you plan to do.

```
Cmd> spool("spool.txt") # start spooling on file 'spool.txt'

Cmd> getseeds() #display current seeds, see Sec. 2.13
Seeds are  1657851609 and   960912167
(1)   1.6579e+09   9.6091e+08

Cmd> x <- runi(5); x # this will be transcribed on spool.txt
(1)      0.67374      0.13423      0.82378      0.89615      0.66544

Cmd> spool() # suspend spooling
Spooling on spool.txt suspended

Cmd> x-3 # this will not be transcribed on spool.txt
(1)      -2.3263      -2.8658      -2.1762      -2.1038      -2.3346

Cmd> spool() # resume spooling.  This line will not be transcribed
Resume spooling on spool.txt

Cmd> x+1 # this line and following output will be transcribed
(1)       1.6737       1.1342       1.8238       1.8962       1.6654
```

This is what ends up in file `spool.txt`.

```
Cmd> getseeds() #display current seeds, see Sec. 2.13
Seeds are  1657851609 and   960912167
(1)   1.6579e+09   9.6091e+08

Cmd> x <- runi(5); x # this will be transcribed on spool.txt
(1)      0.67374      0.13423      0.82378      0.89615      0.66544

Cmd> spool() # suspend spooling
Spooling on spool.txt suspended

Cmd> x+1 # this line and following output will be transcribed
(1)       1.6737       1.1342       1.8238       1.8962       1.6654
```

In the windowed versions of MacAnova, it is also possible directly to save the contents of the command window to a file using **Save Window** or **Save Window As...** on the **File** menu. See Appendices B, D and F.

2.17 **Using save() and restore() to preserve work between sessions** All the variables and macros you are currently using are referred to collectively as the *workspace*. During a MacAnova run, the workspace is in computer memory (RAM) and not in a file on disk. When you quit MacAnova, unless you take precautions, your workspace and often the result of a lot of work is normally lost. You can save everything in your workspace on

disk by save(fileName), where fileName is a quoted string or CHARACTER variable specifying the name of the file. On a later MacAnova session, restore(fileName) restores the workspace to the way it was when saved. As usual in windowed versions, the empty file name "" lets you select a file.

```
Cmd> x # show that vector x is defined
(1)      0.67374      0.13423      0.82378      0.89615      0.66544

Cmd> save("session1.sav") # save workspace, including vector x
Workspace saved on file session1.sav

Cmd> quit  # terminate the run
```

Now suppose you start a new MacAnova session.

```
Cmd> x # At this point x is not defined
UNDEFINED

Cmd> restore("session1.sav") # restore the save file
Restoring workspace from file session1.sav
Workspace saved Thu Nov 30 11:52:58 1995

Cmd> x # variable x as we previously saved it is now defined
(1)      0.67374      0.13423      0.82378      0.89615      0.66544
```

The file created by save() is in a binary format readable only by MacAnova running on the same type of computer. It is useless on any other type of computer. In contrast, asciisave(fileName) saves your workspace in a form that can be read by MacAnova on any type of computer. The file it writes is a plain text or ASCII file although it has an arcane format intended to be read only by a computer. You can even safely send it via e-mail with little danger of corruption since there are no non-printable characters in the file. Command restore() can read either type of file although a file written by asciisave() may take a little longer to restore. The file created by asciisave() is often larger than the one created by save(), although that need not be the case.

Another use for save() and restore() is when you are running MacAnova on an unstable computer, subject to crashes, or running a Unix version over the telephone using a modem where there is a chance the connection will be broken. If you use save() every few minutes, you minimize the danger of losing work. To make this easier, after the first use of save(fileName), simply save(), omitting fileName, saves to the same file. If the original saving was done by asciisave(), a plain text files will be written.

To save your workspace on a windowed version, select **Save Workspace** on the **File** menu or press ⌘**K** on a Macintosh or Control-K in the Windows or Motif versions. The first time you do this, you will be prompted for a save file. Later uses of **Save Workspace** will save on the same file with no prompting.

See Sec. 7.7 for information on partial saves and keywords all and delete.